Autotuning wavefront applications for multicore multi-GPU hybrid architectures

Autotuning Wavefront Applications for Multicore Multi-GPU Hybrid Architectures

Siddharth Mohanty
Institute for Computing Systems Architecture
University of Edinburgh, UK
s.mohanty@sms.ed.ac.uk

Murray Cole
Institute for Computing Systems Architecture
University of Edinburgh, UK
mic@inf.ed.ac.uk

ABSTRACT

Manual tuning of applications for heterogeneous parallel systems is tedious and complex. Optimizations are often not portable, and the whole process must be repeated when moving to a new system, or sometimes even to a different problem size. Pattern-based programming models provide structure which can assist in the creation of autotuners for such problems. We present a machine learning based auto-tuning framework which partitions the work created by applications which follow the wavefront pattern across systems comprising multicore CPUs and multiple GPU accelerators. The use of a pattern facilitates training on synthetically generated instances. Exhaustive search space exploration on real applications indicates that correct setting of the tuning factors leads to a maximum of 20x speedup over an optimized sequential baseline, with an average of 7.8x. Our machine learned heuristics obtain 98% of this speed-up, averaged across range of applications and architectures.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Design Studies; D.1.3 [**Programming Techniques**]: Concurrent Programming-*Parallel programming*

Keywords

wavefront pattern, auto-tuning, multi-GPU

1. INTRODUCTION AND BACKGROUND

The advent of heterogeneous systems comprising multicore CPUs and manycore accelerators such as GPUs, has increased the computational power available to everyday users, but has come at a price to the application developer and programming toolchains. The developer now has to navigate diverse languages and libraries, and integrate these within single applications. Performance tuning of such applications is more complicated than tuning essentially homogeneous systems. Finding a programming methodology and toolchain which can address these challenges is widely recognized as being of major importance, both academically and industrially [5].

Pattern-oriented parallel programming [12] offers a promising approach to the heterogeneous parallelism challenge, by encapsulating parallel decomposition and distribution behind an API which requires the programmer to code only application specific aspects. This approach not only simplifies the programmer's task but also presents the system with a constrained optimization challenge of choosing between and tuning parameters of a set of candidate, heterogeneous parallelizations. This can provide a basis for performance portability. We present a case study in the application of this approach. Our selected pattern is the *wavefront*. Our implementation strategy distributes wavefront applications across systems which incorporate a multicore CPU and multiple GPU accelerators. In order to better understand the tuning tradeoffs, and to assist in the evaluation of our heuristics, we have performed an exhaustive exploration of an interesting fragment of the tuning space, across a collection of systems comprising a CPU and single or multiple GPUs. Since such an exhaustive search would be impractical in a production system, we have investigated the application of machine-learning strategies to reduce the search time. We have experimented across a range of wavefront applications and heterogeneous systems. The wavefront pattern [6] abstracts computations which evaluate a class of multidimensional recurrence relations. Figure 1 gives a graphical representation of a two-dimensional wavefront. The values of the relation are computed into a multidimensional array. Computation starts at position (0,0) and propagates to neighboring elements in a series of diagonal bands, resulting from the dependencies inherent in the pattern. This wave-like sweep of computation gives the pattern its name.

For our purposes, the key characteristics of a wavefront instance are as described in table 1. *dim* is the number of rows in the array. For simplicity we assume square arrays, but this restriction could be lifted straightforwardly. *tsize* captures the granularity of the computation at each point in the array, which we assume to be regular as typically the

Parameter	Description
dim	width of the array
tsize	granularity of the element computation
dsize	element data size

Table 1: Input Parameters

PMAM'14 February 15–19, 2014, Oralando, FL, USA

http://dx.doi.org/10.1145/2560683.2560689.

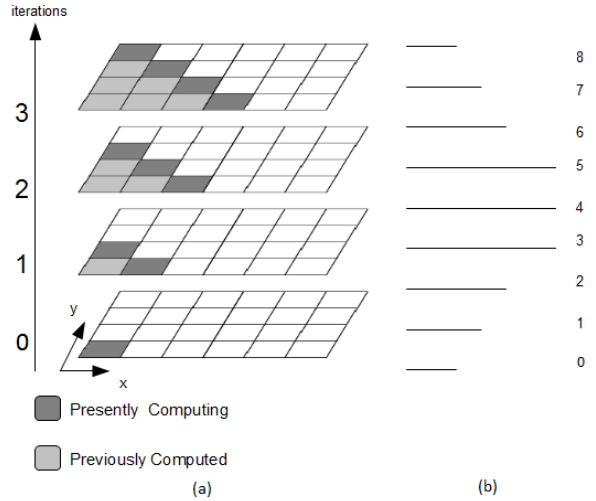

Figure 1: (a) Waveflow for a two dimensional instance of size 4 x 6 (b) The number of concurrently computable elements increases from iteration 0 until maximum parallelism is achieved at iterations 3,4 and 5. Part (b) of the figure is inspired by [1].

case. *dsize* refers to the number of floating point data items at each point in the array, providing a measure of data granularity. These characteristics will form the input parameters to our autotuning framework. Their experimental values will be discussed in section 3.1.1.

The remainder of this paper is organized as follows. Section 2 presents our implementation strategy, its tuning points and the trade-offs these create. Section 3 discusses our experimental programme, covering the applications considered, implementation space and overall autotuning strategy. Section 4 discusses the results of our exhaustive evaluation of the tuning space. Section 4.2 evaluates our machine learning strategies used for autotuning. We review related work in section 5 and present our conclusions and future work in section 6.

2. IMPLEMENTATION STRATEGY

Our parallel wavefront execution strategy extends previous work [3] with support for GPU tiling and the use of multiple GPUs.

In a wavefront, data point computation time is roughly homogeneous so maximum parallelism occurs at the diagonal. Within a diagonal, computation of each data point is independent, hence overall diagonal computation is data parallel. The Single Instruction Multiple Thread (SIMT) constraints of the GPU architecture are thus satisfied by the diagonal major representation of data and successive diagonals can be offloaded onto a GPU. However, it is intuitively clear that this is only beneficial for diagonals of sufficient size and/or computational granularity to amortize the costs of transferring data to and from the device and of initializing execution. Determining these diagonals is a machine and application dependent tuning criterion. For the remaining data points, CPU computation is preferable and it is a common optimization to partition this space into rectangular tiles, computing all points in a tile sequentially in order to benefit from cache re-use. Optimal selection of tile size is also machine and problem dependent [10, 13].

Tiling within a GPU [1], reduces global memory access within the GPU and leads to local cache reuse, besides invoking fewer kernel calls from the host CPU. GPU tiles map to work-groups in OpenCL and the elements within the tile map to work-items or GPU threads. Within a work group, the work items have to be synchronized to follow the wavefront pattern. This introduces an overhead. The GPU tile size (our *'gpu-tile'*) tunable parameter is restricted by hardware and problem size.

Our single-GPU parallel implementation strategy therefore has three phases and three tunable parameters - number of diagonals to offload onto a GPU (or *'band'*) and the tile size of CPU and GPU(*'cpu-tile'* and *'gpu-tile'*). In the first phase, tiled parallel computation proceeds using all cores of the CPU. In the second phase, execution switches to the GPU where it proceeds, possibly tiled, diagonal by diagonal. In the third phase, computation reverts to the CPU and is completed in tiled parallel fashion. This implementation strategy is illustrated in the figure 2. The second phase,

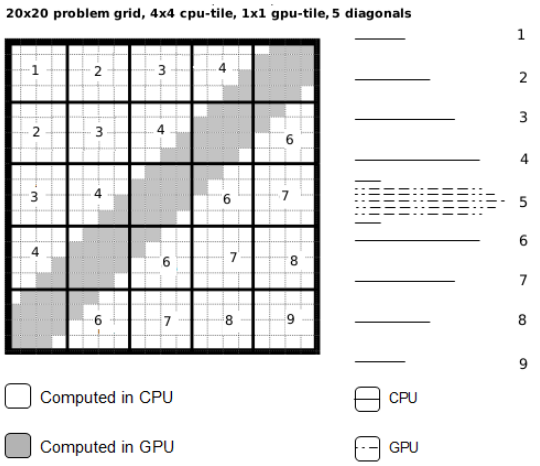

Figure 2: Implementation strategy showing three phase computation for 20 x 20 grid. Phase 1 and 3 have CPU tiles of size 4x4 and phase 2 is GPU consisting of its 1D work groups, with each kernel call corresponding to one diagonal

or in principle the first and third phases, may be null. In the latter case, computation is carried out entirely within the GPU.

The presence of multiple GPUs introduces two further tuning parameters. We must decide how many GPUs to exploit (tuning parameter *gpu-count*). Furthermore, partitioning data among multiple GPUs is non trivial and communication among GPUs is expensive. Wavefront dependencies force data in the border regions (or *'halo'*) of partitioned diagonals to be shared among the GPUs. This is shown for two GPUs in figure 3. As successive partitioned diagonals within each GPU get computed, their border data becomes stale. This necessitates halo exchanges (or *'swaps'*) between the neighbouring GPUs, depending on the extent of overlap or *halo* size. Each time this happens, data elements have to be first transferred to the host (CPU) memory and then transferred to respective destination GPUs. The overhead

from data communication mandates minimising communication between GPUs. However increasing *halo* size causes more redundant computation. Thus *halo* size is our fifth tunable parameter. To summarise, the tunable parameters in

Parameter	Description
cpu-tile	side length of the square tiles for CPU tiling
band	number of diagonals on each side of the main diagonal, to be computed on the GPU
gpu-count	number of GPU devices to use
gpu-tile	the GPU equivalent of CPU tiling
halo	size of the halo for dual GPUs

Table 2: Tunable Parameters

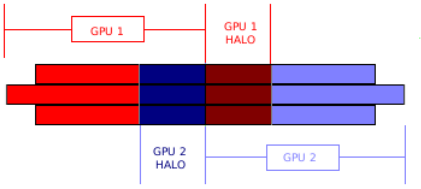

Figure 3: The partitioning of three diagonals among two GPUs with subsequent halo regions

our implementation strategy are as listed in table 2. These will be the targets of our autotuning framework. In the next subsection we discuss tuning trade-offs. The tunable three phase strategy itself is captured in our library code, using threads to control CPU phases and our own OpenCL harness to control communication with and execution upon the GPU.

2.1 Performance tuning trade-offs

For the wavefront pattern, GPU computation becomes feasible when there is enough parallelism to be exploited. Thus a) the problem size (*dim*) should be large enough, since smaller sized problems can be computed quicker in the faster CPU cores and b) the granularity of task (*tsize*) should be high so that computation dominates over the cost of starting a GPU and the communication overhead of transferring data between GPU and CPU. This communication cost naturally increases when data size (*dsize*) being transferred increases. Another factor that increases communication cost is the number of GPUs employed. While with a single GPU data is transferred from/to CPU only twice, dual GPUs have the additional overhead of exchanging neighbouring data between themselves every few iterations (*halo* swapping). This overhead becomes more expensive if the data size is large as more time is spent in swapping halos. A reduction in halo swaps is obtained by increasing the *halo* size. The diagonal major structure of the problem grid in the GPU restricts this *halo* size to a maximum of the length of the start/end diagonal. Even at maximum size, the advantage gained from fewer swaps has to be traded against redundant computation, which starts affecting performance with increasing granularity of task.

Communication cost is also affected by tiling (*gpu-tile*) the GPU since this reduces the number of kernel calls required but incurs the additional cost of synchronizing work items within each work group. If computation dominates over communication anyway, time spent in kernel calls no longer matters and tiling would then prove to be counter productive.

Finally, the type of system affects the performance - a fast GPU coupled to a slow CPU means data will mostly be offloaded to the GPU (unless bandwidth is the bottleneck) leading to higher values of *band*. In such a system, CPU tiling will have negligible effect as most of computation is carried out in the GPU. Likewise, in fast CPU-fast GPU systems, good *band* values will be correspondingly lower.

3. EXPERIMENTAL PROGRAMME

We now describe our experimental programme. Our overall strategy is presented in figure 4, and is line with standard applications of machine learning to the tuning of computer systems [11]. Our goals are to understand the relationship between settings of the internally tunable implementation parameters and performance, and to use machine learning techniques to control the automatic setting of these parameters. The first phase of our experimental programme deals with training our model, using the synthetic wavefront application. The second phase applies the learned model to real, previously unseen wavefront applications.

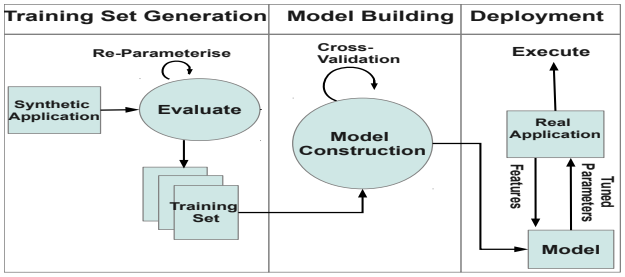

Figure 4: Machine Learning Strategy : The training set is created by selecting high performing instances from an exhaustive parameterized search of the synthetic wavefront application. Decision tree models are built from the training set and cross validated. In deployment, the model is passed features of the previously unseen application and returns appropriate tuning parameter settings.

3.1 Training Phase

Training is conducted with a synthetically generated wavefront application. This is parameterizable across a wide range of size and granularities. It is a strength of the pattern-oriented approach that such an approach is feasible, removing the need to find real applications for the training phase.

3.1.1 Parameter Space

In order to gain insights into the shape of the performance space and trade-offs, we first conduct an exhaustive evaluation of our synthetic application, across a range of settings for the input and output parameters, as listed in table 3.

dim is straightforward. *tsize* is measured in units of the execution time of a single iteration of the synthetic kernel function on a single CPU core. The data structure for each element in our synthetic application consists of two int vari-

Parameter	Range
dim	500 to 3100
tsize	10 to 12000
dsize	1, 3, 5
cpu-tile	1, 2, 4, 8, 10
band	-1 to 2**dim*-1
gpu-count	0, 1, 2
halo	-1 to 0.5*(length of first offloaded diagonal)
gpu-tile	1, 4, 8, 11, 16, 21, 25

Table 3: Parameter Ranges

ables and a varying number of floats, controlled by *dsize*. For example, *dsize=5* means size of each element is 8+5∗8=48 bytes and so on.

Values of parameters like *dim, tsize, band, halo* are spaced irregularly to avoid any cyclic pattern and incorporate a degree of randomness as later the best performing values are used in training our learning models.

To simplify modelling, we have overloaded the *band* and *halo* parameters to encode *gpu-count*. Thus, since a *band* of n means that $2n + 1$ diagonals in total are assigned to the GPU, a *band* of -1 means that the GPU is not to be used. Larger band values mean that at least one GPU is used, with a non-negative *halo* size meaning that the *gpu-count* is 2.

To enable us to explore the parameter space within a reasonable time, we set a threshold limit of 90 seconds on the runtime *rtime* for any execution. This has no impact on our tuning since any point that exceeds this threshold limit is already a very bad configuration which would not be selected as a training example. We removed the threshold in collecting points for our serial baseline in order to correctly compute performance improvement.

3.1.2 Autotuning Strategies

We used decision trees to derive our learning model, using training data drawn from the synthetic application. Training sets are created by subsetting the exhaustive search data as follows: firstly a subset of the problem instances (i.e., by *dim*, *tsize* and *dsize*) are selected by regular sampling; then the best five performance points for these instances (by tunable parameter values) are added to the training set. The intuition is that these should be representative of the good decisions we wish to embed in our models. Initial evaluation is done through *cross-validation*, meaning evaluation is conducted on instances of synthetic application which were omitted from the training set at the first step, to avoid overfitting. We explored different configurations of the learning model to obtain test results that were at least 90% accurate. This model was then applied to the real applications. This procedure is repeated independently for each system, in line with a scenario which would see the software trained "in the factory".

During training, we first build a binary SVM based predictor to decide whether or not to exploit parallelism. For those cases in which parallelism is predicted to be beneficial we then apply and evaluate two machine learning heuristics, based on M5P Decision Tree and REP Tree [9]. Previous work [3] found simple Linear Regression models lacking, and upon exploring different learning models we found the decision trees to be most accurate in predicting optimal values for our tunable parameters.

3.2 Evaluation Phase

We evaluated the performance of our learned model on two real world wavefront applications. These two applications are summarized below.

3.2.1 Evaluation Application Suite

Nash Equilibrium [15] : A game-theoretic problem in economics, characterized by small instances but a very computationally demanding kernel. The internal granularity parameter controls the iteration count of a nested loop.

Biological Sequence Comparison [2] : A string alignment problem from Bioinformatics, characterized by very large instances and very fine-grained kernels, varying with detailed comparisons made.

The input parameter values of these real world applications map to our synthetic scale as follows: one iteration of Nash corresponds to a *tsize*=750 with data granularity of *dsize*=4, while the Biological Sequence Comparison application has *tsize*=0.5 and *dsize*=0.

3.3 Platforms

Our three experimental systems are described in a table 4. 'HT' stands for hyper-threaded CPU cores and 'CU' refers to the GPU compute units.

System	Freq (Mhz)	Cores (HT)	Mem (GB)	GPU	Freq (Mhz)	CU	Mem (GB)
i3-540	1200	4	4	GTX 480	1401	15	1.6
i7-2600K	1600	8	8	4x(GTX 590)	1215	16	1.6
i7-3820	3601	8	16	Tesla C2070, C2075	1147	14	6.4

Table 4: Experimental Systems

We measure runtime of the whole program execution using wall clock timers in the host program, averaging across three runs (which exhibited low variance of less than .01).

4. RESULTS AND ANALYSIS

In section 4.1 we investigate the characteristics of the search space created by our synthetic training application, and explore the resulting model. In section 4.2 we evaluate the model on real world applications.

4.1 Training : Exhaustive Search Results

We now present the results of our exhaustive search space exploration of the synthetic application across all three systems.

4.1.1 Optimal performance points

Figure 5 presents a set of four heatmaps for the two multiple GPU systems and two heatmaps for the single GPU system, with all maps having *tsize* and *dim* as axes, and plotting the values of *band* and *halo* (for multi GPU systems) that result in the fastest execution time. The upper half heat maps correspond to *dsize*=1 (element size=16 bytes) and lower half with *dsize*=5 (element size=48 bytes). From

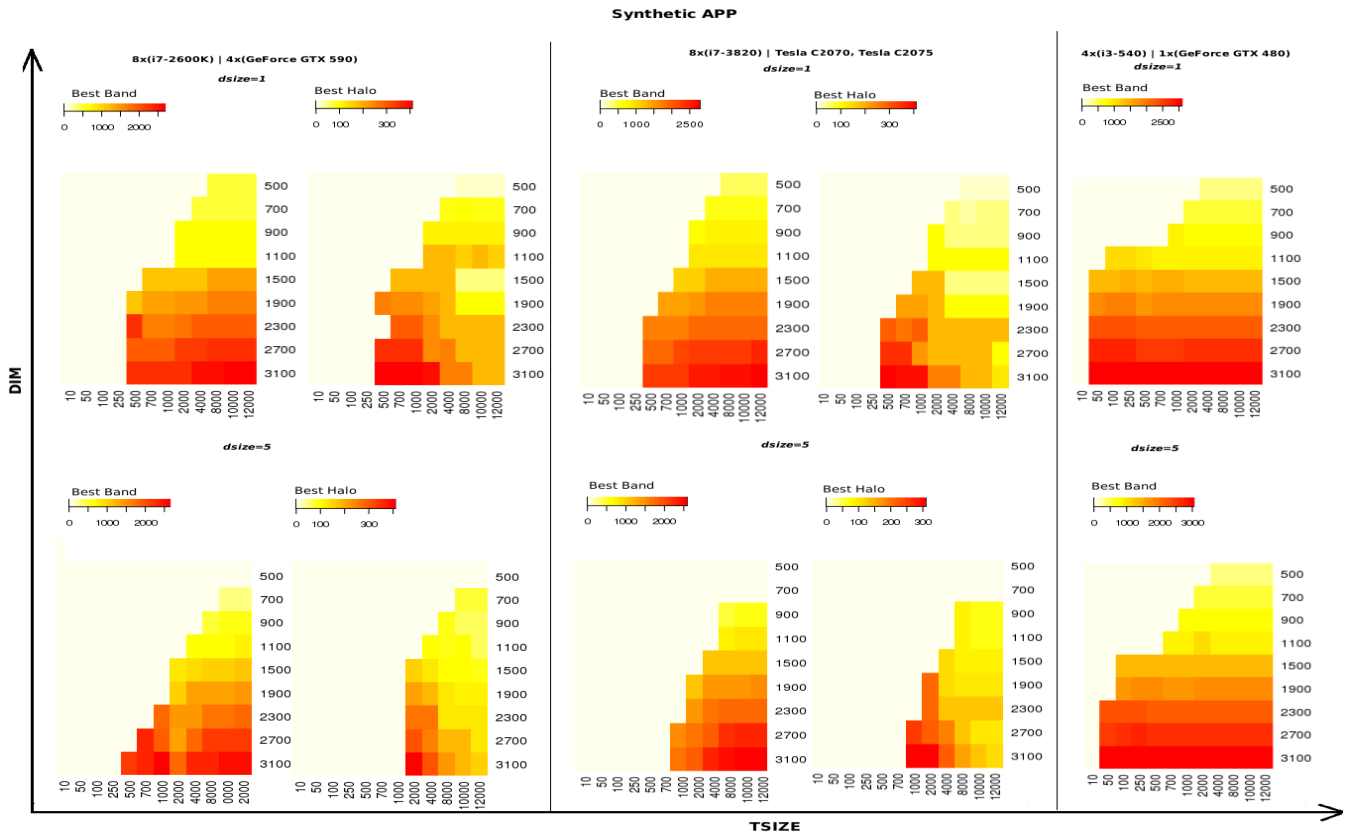

Figure 5: Heatmaps illustrate the *band* and *halo* values at the best performing points from our exhaustive search across three systems and element size of 16 bytes (*dsize*=1; 1 float and 2 ints) and 48 bytes (*dsize*=5; 5 floats and 2 ints). The i3 system is a single GPU system, hence no halo heat map is shown. In all maps the x-axis is *tsize*, indicating kernel task granularity and the y-axis is *dim*, indicating problem size.

the maps it is clear that computing on the GPU becomes favourable (*band*>0) when task granularity exceeds a certain threshold and that this threshold varies depending on the problem size, data size and the hardware. Consider the case of *dsize*=1 (element size=16 bytes) for the i7 systems with fast CPU cores, where the GPU is used from *tsize*≥500 and *dim*≥1900 onwards. This differs from the i3 system with its slower CPU cores where GPU use becomes feasible at a lower threshold of *tsize* ≥ 100 and *dim*≥1100. Apart from the hardware affecting performance parameters, the effect of *dsize* can be seen in all three systems, where the 48 bytes sized elements make GPU use costly as previously discussed, leading to higher thresholds values of *tsize*≥2000 for *dim*≥1900 and *tsize*≥700 for *dim*≥1100 in the i7 and i3 systems respectively. Note that *halo* sizes for the multi-GPU systems are higher when *tsize* values are lower owing to the trade-off between redundant computation cost and lesser communication cost, as discussed in 2.1.

We conclude the heatmap observations by noting that GPU tiling was not beneficial in our search space. This was because tiled GPU performed better than the untiled GPU implementation in cases where the communication costs dominated over computation costs, *tsize* < 50. However in these situations, the CPU only parallel implementation dominated over any GPU based implementation due to the additional overhead incurred from starting the GPU.

4.1.2 *Comparison with simple schemes*

Next we investigate the quality of these heatmap points, by comparing the average speed-up obtained from using these optimal points against the three simple schemes of carrying out computation a) serially in the CPU, b) in parallel across all CPU cores with no GPU phase and c) entirely in the GPU (figure 6).

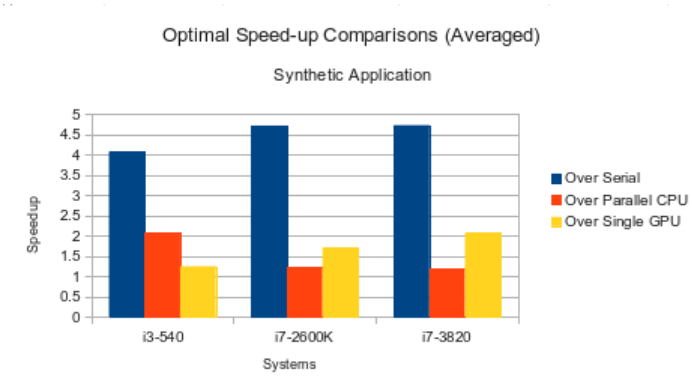

Figure 6: Bars illustrate the speedup of the heatmap points from figure 5 over serial, parallel CPU and single GPU baselines.

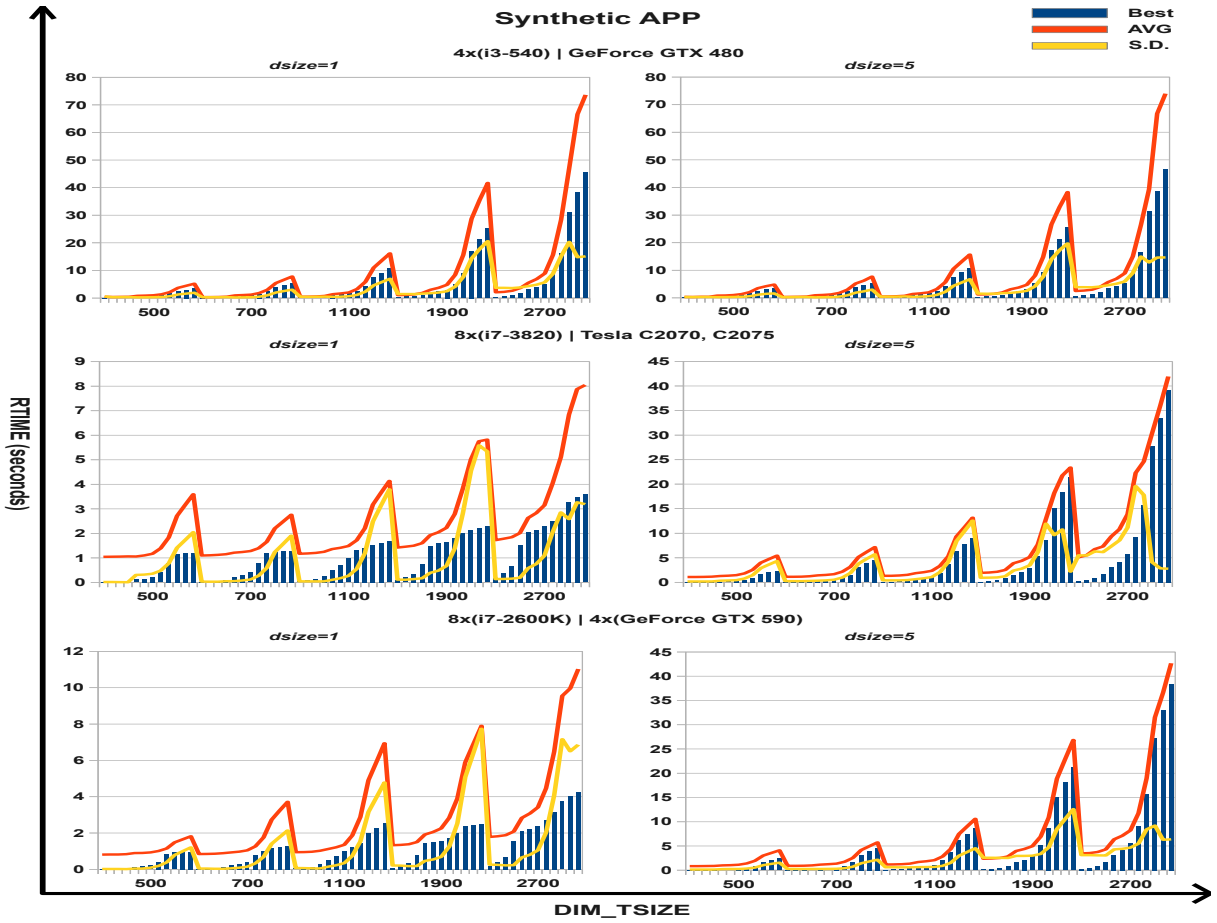

Figure 7: Average case comparison for the Synthetic Application. The x-axis is *dim-tsize*, indicating groups of problem sizes whose kernel task granularity varies from 10 to 12K and the y-axis is *rtime*, indicating actual runtime. Best is the best exhaustive *rtime* (*ber*), AVG is the average *rtime* from all configurations, S.D. is the standard deviation from average. *dsize* refers to the number of floats in our synthetic data structure containing 2 int variables. Total element size = 16 bytes (*dsize*=1; 1 float and 2 ints) and 48 bytes (*dsize*=5; 5 floats and 2 ints)

We note that in case of the i7 systems, on average, doing everything on the GPU, is worse than doing everything on the CPU. This is because the fast CPU outperforms the GPU by a large margin for low task granularity points (up to 10x for $tsize \leq 100$, $dim \leq 1100$).

4.1.3 Average case comparison

The next comparison evaluates optimal heatmap points against average behaviour. This is seen in detail in figure 7, which representing the best exhaustive runtime (abbreviated to *ber*) and the runtime (*rtime*) averaged across all possible combinations of tunable parameters. The figure includes corresponding standard deviations. The x-axis shows groups of *dim-tsize* with *dim* varying 500 to 2700 with each *dim* grouping *tsize* varying from 10 to 12000. The y-axis is the *rtime* in seconds. Both halves show the performance across all three systems when element size=16 bytes and 48 bytes respectively. For *dsize*=1 (element size=16 bytes), the *ber* is 1.5-2 times faster than the average. The standard deviation steadily increases from *dim*=500 to *dim*=1900 due to the widening gap between the best performing and worst performing points. At *dim*=2700 there is a sharp drop as the *rtime* values exceeded our 90 second threshold. These points were excluded from the average. In case of *dsize*=5 (element size=48 bytes), the gap between *ber* and average *rtime* for *dim*=2700 at *tsize*=8K,10K and 12K narrows down to being just 20%. With higher *dsize*, the GPU overheads become larger and more points get excluded for exceeding the threshold.

4.1.4 Sensitivity analysis

We now explore how sensitive the best points are to changes in parameter values. Higher sensitivity would indicate that finding these points is challenging, whereas low sensitivity would indicate that simple random methods might suffice. Owing to space limitations we restrict our discussion of the exact distribution of points to two samples of *dim*=700 and *dim*=2700 belonging to the i7-2600K system. Figure 8 shows violin plots (a combination of box-plots and kernel densities) for these examples. We picked these two samples for *dsize*={1,5} as they are close to the boundary cases in our search space and they conclusively highlight how difference

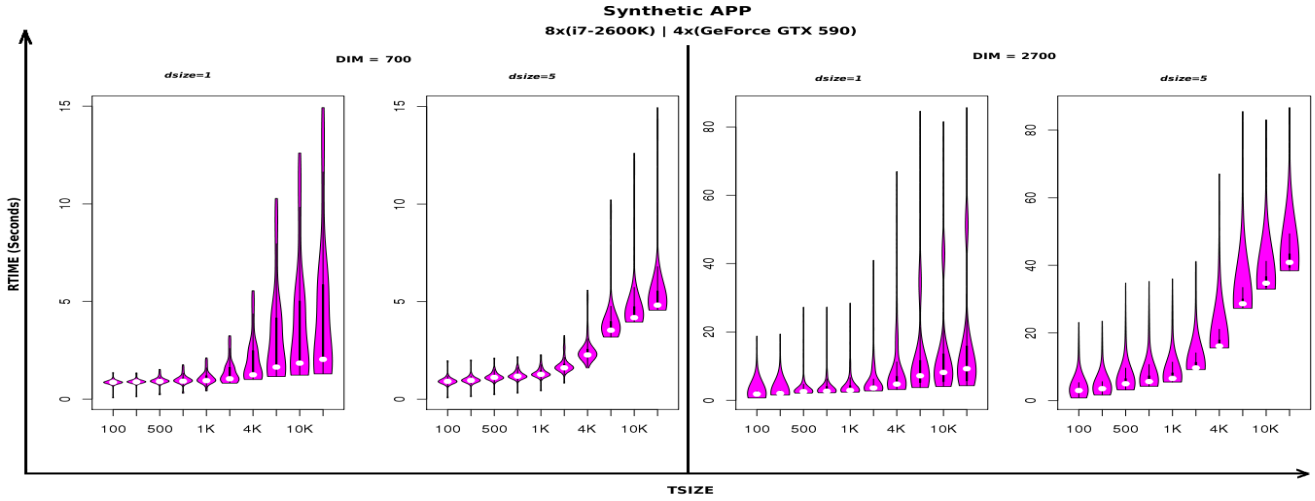

Figure 8: Violin plots showing dispersion of all configurations. The best points are at the base and the white spots are the medians. The x-axis is *tsize*, indicating kernel task granularity and the y-axis is *rtime*, indicating actual execution time.

in problem size and data granularity (and corresponding variation in kernel task granularity within them) impacts the search space. For *dim*=700 we note that most of the points in *tsize*=100 to 1K are dispersed around the median value (represented as the white dot) with the best and worst points at the extreme ends. This is due to the best configuration in these cases being all CPU (see the heatmap in figure 5 showing *band*=-1 for i7-2600K where *dim*=700,*tsize*$\leq$2K). In that case the tunable parameters are only *cpu-tile* and *dsize* resulting in configurations numbering in tens instead of thousands. Contrast this with *tsize*$\geq$2K and for all points in *dim*=2700 where there are many points less than the median value, as seen from the flat base of each violin. These cases correspond to various combinations of the tunable parameters *band*, *halo* and *gpu-tile* in addition to *cpu-tile*. We also observe that in case of *dim*=2700, *dsize*=5 variations in the former three parameters do not affect performance as much as for *dim*=700. This is also confirmed by the lower gap between average *rtime* and *ber* (see figure 7). However selecting the worst points in these cases, such as computing on the CPU only with *band*=-1 when *dim*=2700, *dsize*=1 and *tsize*$\geq$4K, is quite costly (up to 8 times slower). The worst case in these cases are the best points for *dim*=700, *tsize*$\leq$2K. Thus, while variation in tunable parameter values from the best values within a subset of input configurations may not affect performance, it can affect performance in other subsets.

We note that the best points in some subsets were the worst ones in others and vice versa, meaning that any attempt to hand code heuristics for each case quickly becomes impractical. The exhaustive search results vindicate our choice to pursue auto-tuning strategies based on machine learning.

4.1.5 The learned model

A fragment of the learned model which predicts the optimum *halo* values for the i7-2600K system is shown in figure 9. The regression equation (LM1) shows that *halo* depends on other tunable parameters like *band* and *cpu-tile*. This agrees with our intuition as *halo* values are a measure of

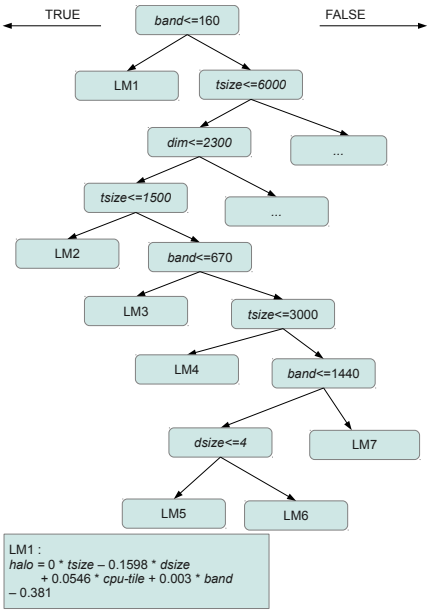

Figure 9: i7-2600K system : The M5 pruned model tree for predicting *halo* values with one linear model (out of 22) shown . As seen, *halo* depends on *band* and *cpu-tile* values, apart from the input parameters of task granularity and data granularity.

the extent of overlap among partitioned diagonals offloaded onto GPUs. Hence, *halo* values depend on *band* values. *cpu-tile* values were predicted using input parameters only (*dim*, *tsize* and *dsize*). This was because on removing other tunable parameters from the regression equations that predicted *cpu-tile* values, accuracy of prediction increased. This also makes intuitive sense since an all CPU configuration has cpu tiling as its only tunable parameter, so other tunable parameters are not needed. *band* values depended on *gpu-tile* values in addition to input parameters. From our exhaustive

search we found *gpu-tile* values corresponded to either 1 or 0 (meaning a GPU was not employed), so it was a binary decision that was accurately predicted using REP Tree. *cpu-tile* and *band* values, like *halo* values, were predicted using the M5 pruned tree model.

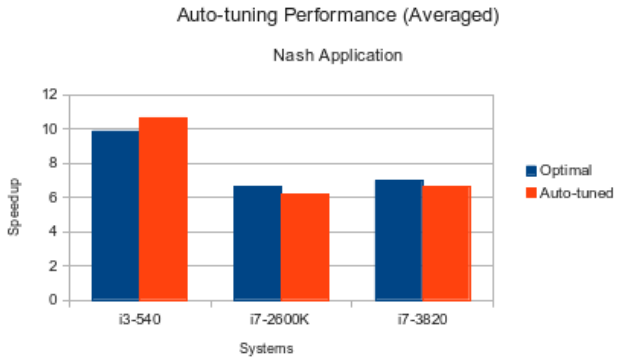

Figure 10: Speedup over sequential baseline from auto-tuning is within 5% of exhaustive search.

4.2 Evaluation : Autotuning Results

For the fine grained Smith-Waterman string compare application autotuning was trivial as the band prediction were 100% accurate, i.e. do everything on the CPU. Our learning model had predicted band=-1 for all $tsize<100$, across our search space of $dim \leq 3100$. Thus in the context of our search space only the predicted *cpu-tile* values differed and selecting the best points was trivial.

A summary of our auto-tuner's performance for the Nash application is shown in figure 10. This figure describes for each system, the average optimal speed-up against a sequential baseline found during exhaustive search of Nash, and the speed-up obtained by our auto-tuner.

The super-optimal performance in the case of the i3-540 is explained by the fact that our regression model based tuner is free to select parameter values which lie outside the set of cases explored in the (necessarily finite) full search. The better quality predictions for the i3-540 can be explained by considering a) it is a single GPU system with only two tunable parameters *band* and *cpu-tile*, i.e. less parameter values to predict as compared to the multi-GPU systems and b) its four CPU cores are slow relative to its GPU, meaning most of the data is often offloaded onto the GPU, easing prediction as compared to the i7 systems with fast CPU cores.

We conclude this section with a detailed visualization of how our auto-tuning fares against the best exhaustive runtime or '*ber*' (figure 11). The *rtime* after autotuning is slightly lower than the *ber* for the i3-540 at many points (as discussed above), while it is slightly higher for the i7 systems as prediction is harder.

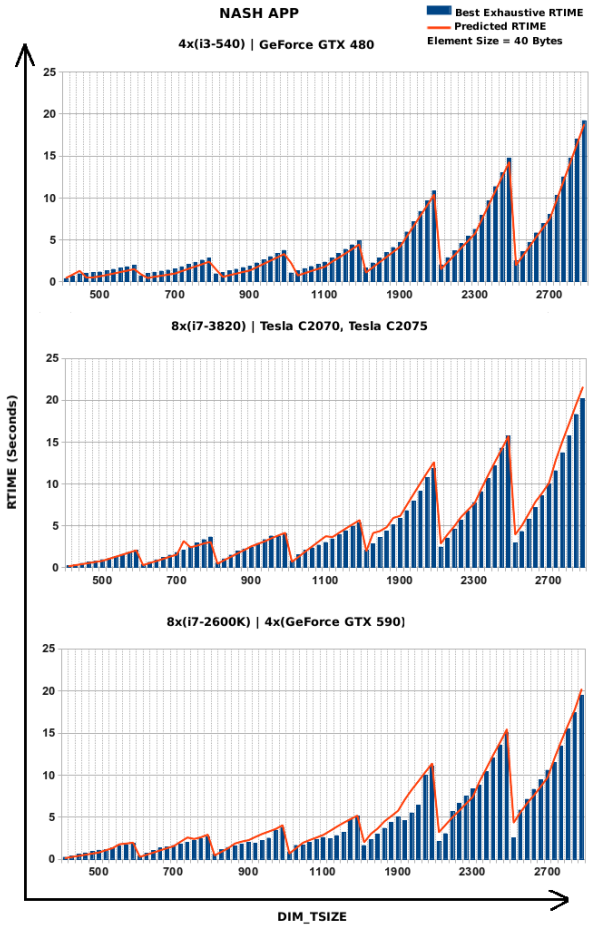

Figure 11: The bars represent runtime of optimal points found from exhaustive search and the line represents runtime from auto-tuning. The x-axis is *dim-tsize*, indicating groups of problem sizes whose task size varies from 10 to 12000 and y-axis is runtime.

5. RELATED WORK

CO_2P_3S [4] is a wavefront framework that generates parallel programs from user supplied methods and data. However, it is restricted to shared memory architectures and does not employ any optimization techniques for any combination of its application dependent properties. The wavefront abstraction in [15] targets multicore and distributed systems. However, its tunable parameters are specific to distributed systems. It also employs processes instead of threads as they are more adaptable to distributed systems but overhead from processes can impact performance.

Stencils have similar issues, but a different dependency pattern to wavefront. Autotuning for the stencil pattern has been widely investigated (e.g. [10, 14, 16]). A multi-GPU framework to handle stencils is covered in [18]. A key difference with our implementation is the absence of dependence between elements in a stencil pattern, which means halo swapping is less frequent for stencils distributed over multiple GPUs than for wavefronts. Dynamic autotuning of multi-GPU/multicore CPU systems can also be based on analytical models [17]. However the problem class considered doesn't belong to the dynamic programming class of problems and auto-tuning is done without resorting to machine learning. Among dynamic auto tuning frameworks, the Active Harmony framework [8] uses the greedy or Nelder Mead algorithm to search a high dimensional space and the tuning results are then treated as a new experience to update the

data characteristics database for future reference. Performance models for wavefront applications on GPU-enhanced HPC systems are presented in [7]. Machine learning techniques have been successfully employed to efficiently explore the CPU-GPU optimization space in [11], though here the decision tree models were used to select either multi-core CPU or GPU implementation and not a hybrid CPU + multi-GPU setup.

6. CONCLUSIONS AND FUTURE WORK

We have presented a framework that successfully encapsulates decomposition and distribution of wavefront computations across CPU cores and GPUs while automatically selecting high quality configurations with respect to problem size, data size and kernel task granularity. We demonstrated that well chosen settings for the number of diagonals to be offloaded (*band*) and length of overlap of computation between GPUs (*halo*) can produce significant improvements in the performance, while tiling inside the GPUs (*gpu-tile*) did not affect performance within our simple search space. Correspondingly, poorly chosen settings resulted in performance which was far from optimal. Our decision tree based auto-tuners were modelled on training data from instances of a synthetic application. This successfully predicted the optimal values for various tunable parameters for the fine grained Biological Sequence Comparison and coarse grained Nash wavefront applications, across three different systems, finding an average of 98% of the performance achieved by an exhaustive search. In future we plan to extend our framework to incorporate other dynamic programming problems, beyond simple wavefronts, such as the 0/1 knapsack problem [19]. We aim to enhance our tiled multi-GPU strategy by incorporating more than two GPUs and plan to upgrade our offline auto-tuner to tune at runtime.

7. REFERENCES

[1] A. M. Aji and W. Feng. *Accelerating Data-Serial Applications on Data-Parallel GPGPUs: A Systems Approach.* TR-08-24, Computer Science, Virginia Tech, 2008.

[2] C. Alves, E. Cáceres, F. Dehne, and S. Song. A parallel wavefront algorithm for efficient biological sequence comparison. *ICCSA'03*, pages 249–258, 2003, Springer-Verlag.

[3] S. Mohanty and M. Cole. Autotuning wavefront abstractions for heterogeneous architectures. In *Third Workshop on Applications for Multi-Core Architectures, 2012*, WAMCA '03 pages 42–47, New York, NY, USA, 2012. IEEE.

[4] J. Anvik, S. Macdonald, D. Szafron, J. Schaeffer, S. Bromling, and K. Tan. Generating parallel programs from the wavefront design pattern. In *7th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 1–8. Society Press, 2002.

[5] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *CACM*, 52(10):56–67, 2009.

[6] The Wavefront pattern. *www.cs.uiuc.edu/homes/snir/PPP/patterns/wavefront.pdf.*

[7] S. D. Hammond, G. R. Mudalige, J. A. Smith, and S. A. Jarvis. Performance prediction and procurement in practice: Assessing the suitability of commodity cluster components for wavefront codes. *IET SOFTW.*, 3(6):509–521, 2009.

[8] J. K. Hollingsworth and P. J. Keleher. Prediction and adaptation in active harmony. *Cluster Computing*, 2:195–205, July 1999.

[9] E. F. Ian H. Witten. *Data Mining: Practical Machine Learning Tools and Techniques.* Morgan Kaufmann, 2005.

[10] S. Kamil, C. Chan, S. Williams, L. Oliker, J. Shalf, M. Howison, and E. W. Bethel. A generalized framework for auto-tuning stencil computations. In *Proceedings of the Cray User Group Conference*, 2009.

[11] D. Grewe, Z. Wang, and M. O'Boyle. Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems. In *Proceedings of the 11th International Symposium on Code Generation and Optimization*, CGO'13, 2013.

[12] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation.* Morgan Kaufmann, 2012.

[13] G. Rivera and C.-W. Tseng. Tiling optimizations for 3d scientific computations. In *2000 ACM/IEEE conference on Supercomputing* , Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[14] M. Christen, O. Schenk and H. Burkhart. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures, *Proceedings of IPDPS 2011*, pages 676-687, IEEE Press, 2011.

[15] L. Yu, C. Moretti, A. Thrasher, S. Emrich, K. Judd, and D. Thain. Harnessing parallelism in multicore clusters with the all-pairs, wavefront, and makeflow abstractions. *Cluster Computing*, 13:243–256, September 2010.

[16] Y. Zhang and F. Müller. Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. *Proceedings of the Tenth International Symposium on Code Generation and Optimization,* pages 155-164, ACM Press, 2012.

[17] M. Boratto, P. Alonso, D. Giménez, M. Barreto, and K. Oliveira. Auto-tuning methodology to represent landform attributes on multicore and multi-gpu systems. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '13, pages 125–132, New York, NY, USA, 2013. ACM.

[18] T. Lutz, C. Fensch, and M. Cole. Partans: An autotuning framework for stencil computation on multi-gpu systems. *ACM Trans. Archit. Code Optim.*, 9(4):59:1–59:24, Jan. 2013.

[19] The Knapsack Problem - an Introduction to Dynamic Programming *http://www.cs.rit.edu/ zjb/courses/800/lec7.pdf*